J2SE 5

Alcune novità introdotte da J2SE5

Seminario di Programmazione ad Oggetti

Lucidi di Luca Ferrari

Outline

- Autoboxing
- Foreach
- Enum
- Varargs
- Output formattato
- Generics
- Ergonomics e performance

Autoboxing-Outboxing

Le classi wrapper

- Java non è un linguaggio completamente ad oggetti
- I tipi primitivi (`int`, `float`, `double`, ...) non sono oggetti!
- Per supportare la gestione dei tipi primitivi uniformemente a quella degli altri oggetti, fin da Java 1 sono presenti delle classi ***wrapper***

Le classi wrapper

- **L'utilizzo delle classi wrapper risulta scomodo il più delle volte:**

```
Vector v = new Vector();
int i = 10;
...
  Integer iObject = new Integer(i);
  v.add(iObject);
...
  iObject = (Integer)v.get(0);
  i = iObject.intValue();
```

Autoboxing - Outboxing

- **Ora è possibile utilizzare direttamente uno scalare in un contesto di oggetto, lasciando che sia il compilatore a gestire il wrapping del tipo primitivo:**

```
Vector v = new Vector(10);

for(int i=0;i<5;i++){
   v.add(i);
   v.add(((float)i)*2.5);
}

for(int i=0;i<v.size();i++){
 System.out.println(i+"="+
                v.elementAt(i));
}
```

output di esecuzione

```
0=0
1=0.0
2=1
3=2.5
4=2
5=5.0
6=3
7=7.5
8=4
9=10.0
```

foreach

foreach

- **È ora supportato il ciclo foreach tramite una sintassi particolare di `for`:**

```
for( variable_type variable_name : list )
```

- **Un ciclo foreach consente di scorrere tutti gli elementi di una lista (ad esempio un array) gestendo automaticamente l'incremento di eventuali indici e l'assegnamento della variabile di ciclo all'elemento successivo**

foreach: esempio

```
public class foreach{
  public static void main(String argv[]){
    String array[] = new String[10];
    for(int i=0;i<array.length;i++){
     array[i] =
          new String("Stringa n."+i);
    }

    // utilizzo del foreach
    for(String s: array){
     System.out.println(s);
    }
  }
}
```

output di esecuzione

```
Stringa n.0
Stringa n.1
Stringa n.2
Stringa n.3
Stringa n.4
Stringa n.5
Stringa n.6
Stringa n.7
Stringa n.8
Stringa n.9
```

foreach: lavorare con strutture dati

```
impor
publ
    {
        for(int i=0;i<10;i++){
            v.add("String n."+i);
        }

        // uso il foreach
        // ATTENZIONE: uso di object!
        for(Object s: v){
            System.out.println(s);
        }
    }
}
```

ATTENZIONE: si deve usare un `Object` perché un `Vector` memorizza (e restituisce) `Object`.
`foreach` non può effettuare i cast automaticamente! Il problema può essere risolto con *generics*.

foreach solo per estrazioni

- **foreach non può essere usato come ciclo di assegnamento, ma solo di estrazione:**

```
public static void main(String argv[]){
   String array[] = new String[10];
      for(String s: array){
      s = "CIAO";
   }
   for(String s:array){
      System.out.println(s);
   }
}
```

output di esecuzione

```
null
null
null
null
null
null
null
null
null
null
```

foreach: considerazioni

- Tutto il lavoro “sporco” viene svolto dal compilatore
- Non si ha accesso all’indice numerico della lista (non si è a conoscenza della posizione dell’elemento corrente)
- Riduce il rischio di errori banali, come indici fuori dai limiti o cicli annidati sulla stessa variabile
- Non è stata introdotta una parola chiave `foreach` (come in Perl, C-shell,…) per rispetto al *legacy-code*

Preparare una classe per foreach

- **Affinché una collezione di dati sia utilizzabile in un ciclo foreach, occorre che implementi l'interfaccia `Iterable`**

java.lang

Interface Iterable<T>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingQueue<E>, Collection<E>, List<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

---

```
public interface Iterable<T>
```

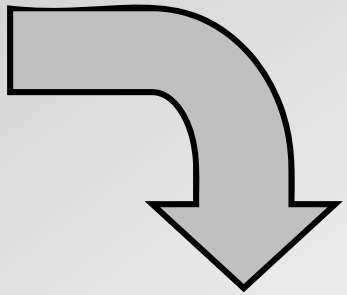
Implementing this interface allows an object to be the target of the "foreach" statement.

Method Summary	
Iterator<T>	**iterator**() Returns an iterator over a set of elements of type T.

foreach: una nota di demerito

- **L'implementazione di foreach non è *null-safe* (al contrario di Perl):**

```
String array[]=null;
for(String s:array){
 System.out.println(s);
}
```



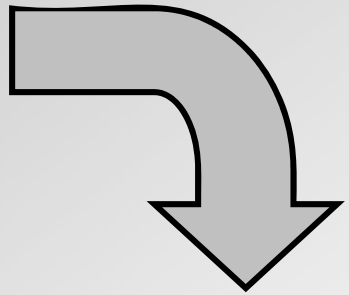
```
String array[]=null;
for(int i=0;i<array.length;i++){
 String s = array[i];
 System.out.println(s);
}
```

```
NullPointerException
```

foreach: una nota di demerito

- **Una implementazione più sicura sarebbe stata:**

```
String arr []=null;
for(String s:arr){
 System.out.println(s);
}
```



```
String arr []=null;
for(int i=0;arr!=null && i<arr.length;i++){
 String s = array[i];
 System.out.println(s);
}
```

varargs

varargs

- È ora possibile definire metodi che accettino un numero variabile di argomenti
- Gli argomenti devono avere tutti lo stesso tipo, al limite `Object` (che equivale a “tutti i tipi possibili”)
- La sintassi prevede l’uso dell’operatore `...` fra il tipo e il nome associato all’argomento
- Simile al concetto di variadic in C/C++, si pensi alla `printf(..)`

varargs: esempio

```
public class varargs {

    public void mvar(String...pars){
        for(int i=0;i<pars.length;i++){
            System.out.println("Parametro "+i+
                                    " = "+pars[i]);
        }
    }

    public static void main(String argv[]){
        varargs v = new varargs();
        v.mvar(“ALFA",“BETA",“GAMMA");
    }

}
```

L’argomento di una funzione varargs è tramutato automaticamente in un array

varargs: esempio equivalente

```
public class varargs {

   public void mvar(String...pars){
     // uso il foreach
       for(String s: pars){
           System.out.println(s);
       }
   }

   public static void main(String argv[]){
       varargs v = new varargs();
       v.mvar(“ALFA",“BETA",“GAMMA");
   }

}
```

varargs: output degli esempi

output di esecuzione

```
Parametro 0 = ALFA
Parametro 1 = BETA
Parametro 2 = GAMMA
```

Versione con ciclo for normale

Versione con ciclo foreach

output di esecuzione

```
ALFA
BETA
GAMMA
```

Overload

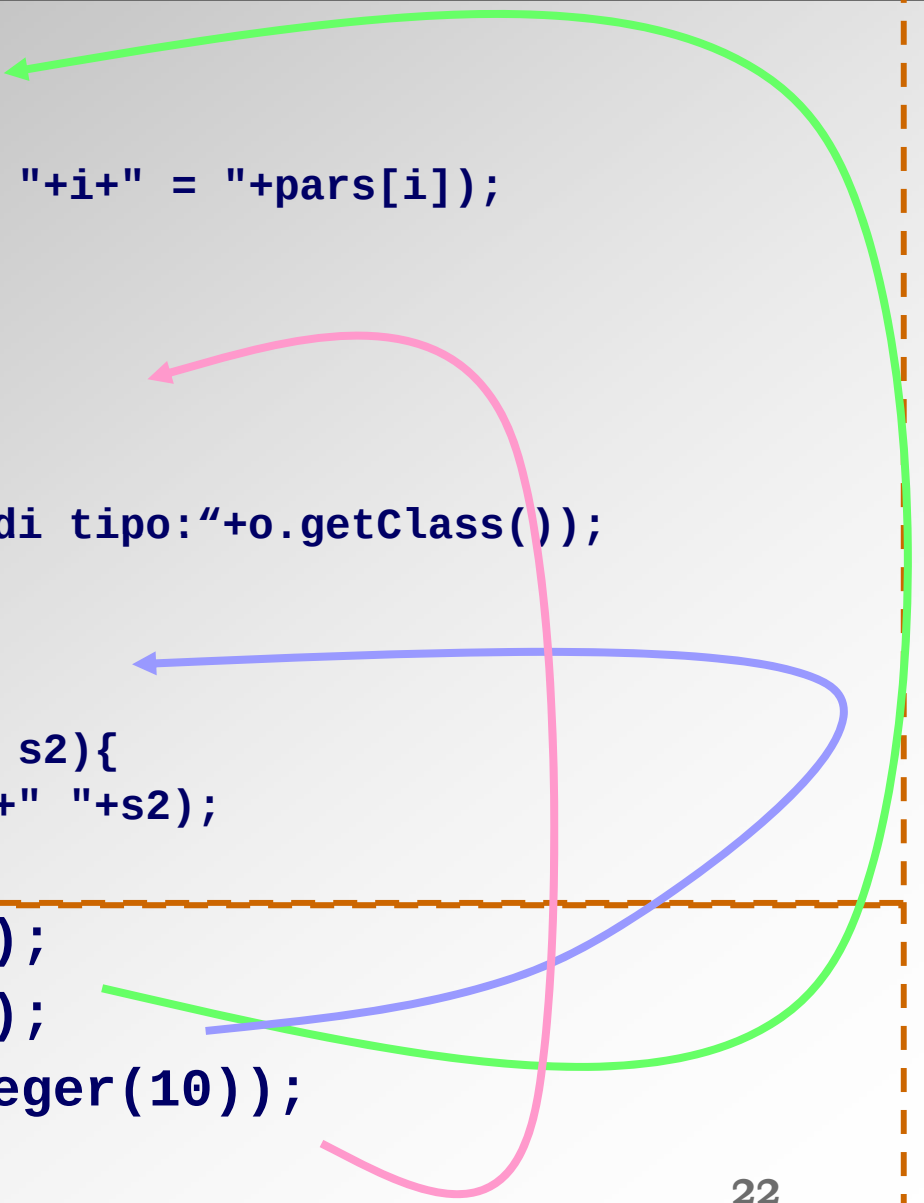
- **È possibile sovraccaricare un metodo varargs con una lista di argomenti dello stesso tipo**

- **Verrà eseguito per primo il metodo il cui prototipo corrisponde perfettamente all'invocazione!**

Overload: esempio

```
public class varargs {
 public void mvar(String...pars){
   for(int i=0;i<pars.length;i++){
      System.out.println("Parametro "+i+" = "+pars[i]);
   }
 }

 public void mvar(Object...pars){
  for(Object o: pars){
     System.out.println("Parametro di tipo:“+o.getClass());
  }
 }

 public void mvar(String s1, String s2){
  System.out.println("stringhe "+s1+" "+s2);
 }
```

```
v.mvar(“ALFA",“BETA",“GAMMA");
v.mvar("Stringa1","Stringa2");
v.mvar(new Object(), new Integer(10));
```



Overload: array

- **Siccome il compilatore tratta i metodi varargs come metodi ad array, il sovraccarico con un array viene impedito:**

```
public class varargs {
  public void mvar(String...pars){...}

  public void mvar(String[] pars){
       for(String s:pars){
           System.out.println("String array: "+s);
       }
   }
}
```

output di compilazione

```
varargs.java:31: mvar(java.lang.String...) is already defined in
fluca.varargs
    public void mvar(String[] pars){
```

enum

Enumerazioni

- **E’ stata aggiunta la parola chiave `enum`, che consente di definire enumerazioni *on the fly***

- **Un utilizzo classico è come lista *static* di valori**

Si presti attenzione a non confondere una enumerazione con una serie di costanti. Seppur simili, una serie di costanti acquisisce importanza in base al valore che si attribuisce alle stesse, mentre una enumerazione ha importanza solo al fine di distinguere gli elementi nella lista.

Enumerazioni: esempio

```
public class EventType{
     public static enum type {
    OPEN,
    CLOSE,
    EXIT,
     };
}
```

Enumerazioni: esempio

```
public class Main1{
    // metodo che accetta un valore enumerato
    public void fireEvent(EventType.type event){
      System.out.println("L'evento ricevuto e'");

      if(event == EventType.type.OPEN)
          System.out.println("APERTURA");
      else
        if(event == EventType.type.CLOSE)
          System.out.println("CHIUSURA");
        else
        if(event == EventType.type.EXIT)
          System.out.println("USCITA");
   }
```

Enumerazioni: esempio

```
public static void main(String argv[]){
     Main1 m = new Main1();
     m.fireEvent(EventType.type.OPEN);
     m.fireEvent(EventType.type.CLOSE);

     System.out.println("Come risulta stampato? -->“

     +EventType.type.OPEN);
     }
}
```

output di esecuzione

```
L'evento ricevuto e'
APERTURA
L'evento ricevuto e'
CHIUSURA
Come risulta stampato? -->OPEN
```

Considerazioni

- **Le enumerazioni sono trattate come oggetti:**

```
$ ls -l
total 36
-rw-r--r--  1 luca users  306 2004-12-09 13:52 EventType.class
-rw-r--r--  1 luca users  121 2004-12-09 13:52 EventType.java
-rw-r--r--  1 luca users 1251 2004-12-09 13:52 EventType$type.class
-rw-r--r--  1 luca users 1200 2004-12-09 14:09 Main1.class
-rw-r--r--  1 luca users  629 2004-12-09 14:09 Main1.java
```

Che tipo di oggetto è un'enumerazione?

```
public void fireEvent(EventType.type event){
  ...
  // analizzo la classe dell'evento
  Class clazz = event.getClass();
  System.out.println("Nome della classe di evento: “
                                  +clazz.getName());
  for(Field f: clazz.getFields()){
    System.out.println("Campo: "+f.getName());
    System.out.println("\t visibilita': ");
    int modificatore = f.getModifiers();
    if(Modifier.isPublic(modificatore)){
      System.out.println("\tpubblico");
    }else if(Modifier.isPrivate(modificatore)){
      System.out.println("\tprivato");
    }
  }
}
```

Che tipo di oggetto è un'enumerazione?

output di esecuzione

```
Nome della classe di evento:
seminario_20.enumerazioni.EventType$type

Campo: OPEN

       visibilita':

      pubblico

Campo: CLOSE

       visibilita':

      pubblico

Campo: EXIT

       visibilita':

      pubblico
```

Output Formattato

L'output formattato

- Le classi di output su stream (ad es. `PrintStream`) di Java hanno il pregio di avere un utilizzo molto semplice ed immediato, ma non consentono un elevato controllo sull'output effettivo

- Altri linguaggi, come ad esempio il C/C++ consentono un accurato controllo dell'output mediante l'uso delle funzioni appartenenti alla famiglia ***printf***

Differenze fra i metodi Java e printf

- **Le differenze fra i metodi `println(..)` di un `PrintStream` Java e la `printf(..)` del C/C++ sono le seguenti:**
  - **`printf` è una funzione *variadic*, `println` no**
  - **`printf` richiede come primo argomento una stringa di formato, seguita dai parametri da convertire in stringa**
  - **`println` consente di inserire il parametro da stampare direttamente nella posizione in cui deve essere stampato (utilizzo dell'operatore + su stringa)**
  - **`printf` ha una implementazione unica indipendentemente dal tipo e numero di argomenti che deve gestire, `println` è sovraccaricato per ogni lista di parametri differente**
  - **`printf` consente di specificare la lunghezza dei valori, `println` richiede l'utilizzo di operazioni di substringing**

Il ritorno della printf

- **Per sopperire alle lacune di formattazione, alcune classi di libreria sono state estese: `String`, `PrintWriter` e `PrintStream` hanno acquisito ciascuna i seguenti metodi (*varargs*):**

```
format(String format, Object... args);
printf(String format, Object... args);
format(Locale locale, String format,
                              Object... args);
printf(Locale locale, String format,
                              Object... args);
```

Esempio: printf

```
public class Main {
    public static void main(String[] args) {
        int    v1 = 10;
        float  v2 = (float)3.69;
        String v3 = "Ciao";
        System.out.printf("valori: v1=%d v2=%f
                                    v3=%s\n",v1,v2,v3);

        String s = String.format("valori: v1=%d v2=%f
                                    v3=%s",v1,v2,v3);
        System.out.println(s);
    }
}
```

output di esecuzione

```
valori: v1=10 v2=3,690000 v3=Ciao
valori: v1=10 v2=3,690000 v3=Ciao
```

Generics

type-safe

- ***Problema:*** **garantire a compile time la coerenza dei dati di una collezione**

- **Il problema, noto come "*type-safe*", richiede la capacità di usare strutture ed algoritmi generali specializzandoli al volo**

Non è banale!

- **Il controllo di tipo avviene sicuramente a run-time, ma potrebbe essere troppo tardi!**

Quando la conformità può creare problemi…

- Mediante la conformità è possibile usare le sottoclassi come fossero classi base
- Ciò produce indubbi vantaggi, consentendo di utilizzare lo stesso algoritmo per tutte le classi derivate
- Si pensi ad esempio alle strutture dati Java (ad es. `Vector`), che sono progettate per accettare qualunque tipo di oggetto…

La conformità può creare problemi di type-safe

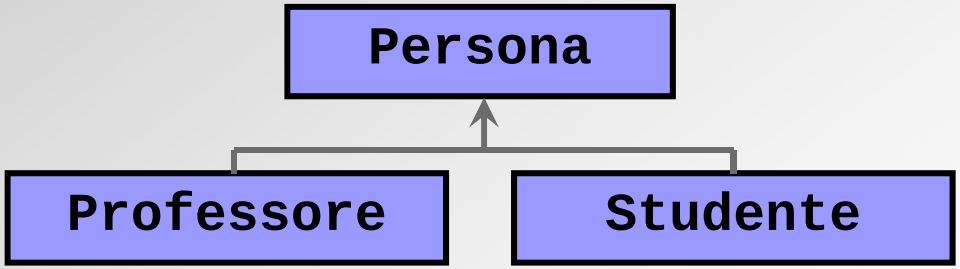
Un esempio concreto

- La soluzione si ha usando i *tipi parametrizzati* (chiamati in Java *generics*)
- Per meglio comprendere, si farà uso del seguente esempio:
  - si dispone di due tipi di persone: studenti e professori
  - si deve costruire un archivio capace di contenere sia studenti che professori
  - l'archivio deve essere omogeneo (non si devono mischiare professori con studenti e vice versa)

Un esempio concreto: implementazione della gerarchia

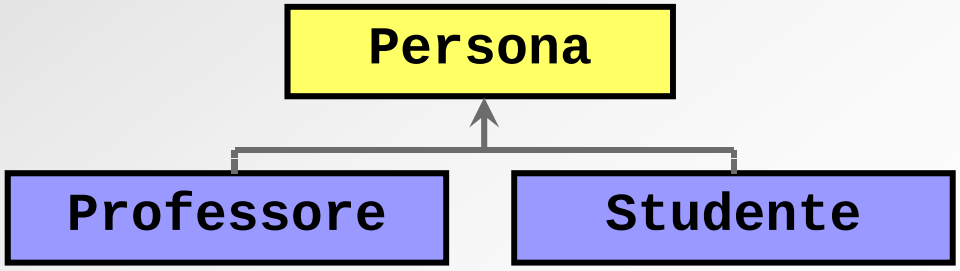
- Studenti e professori sono accomunati dal fatto di essere entrambi persone; inoltre si deve costruire una logica uguale per entrambe le specializzazioni
- Ciò porta alla realizzazione di un legame fra studenti e professori:

una gerarchia!



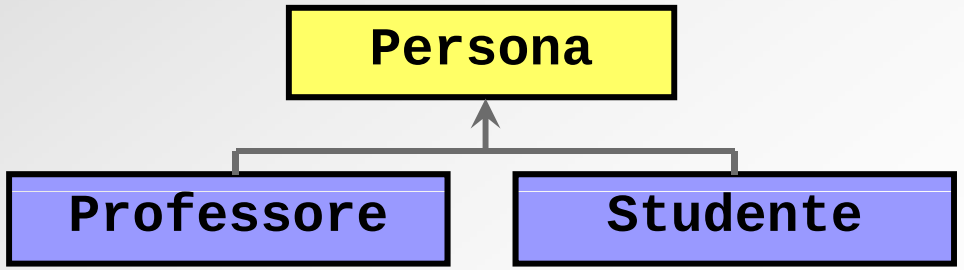
Un esempio concreto: implementazione della gerarchia

```
public class Persona {
    protected String nome, cognome;
    protected int eta;

    public Persona(String nome, String cognome,
                                int eta){
        this.nome = nome;
        this.cognome = cognome;
        this.eta = eta;
    }
```



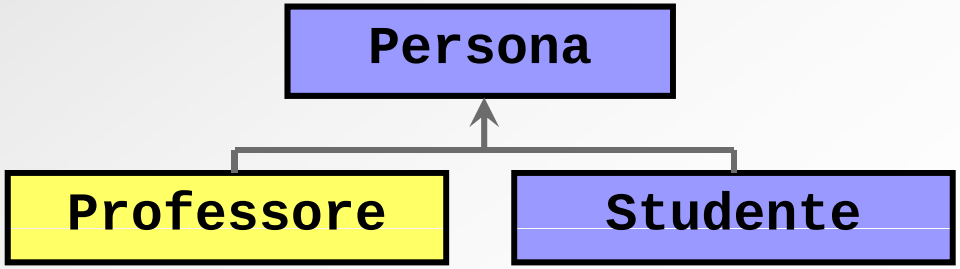
Un esempio concreto: implementazione della gerarchia

```
    public String toString(){
    return nome+" "+cognome+" "+eta;
    }
}  // fine della classe Persona
```



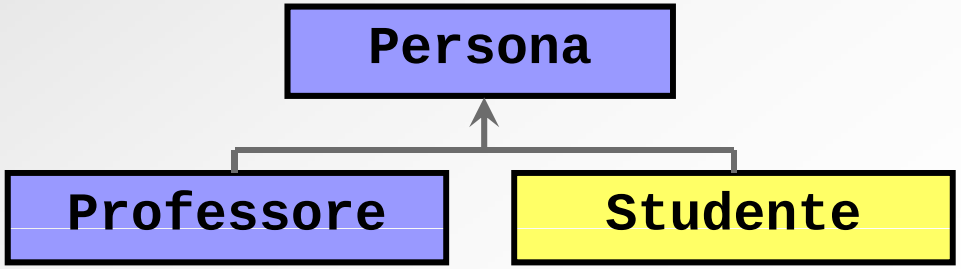
Un esempio concreto: implementazione della gerarchia

```
public class Professore extends Persona{
    public Professore(String nome,
                      String cognome, int eta) {
        super(nome,cognome,eta);
    }

    public String toString(){
      return nome+" "+cognome+" "+eta+
                     " - professore";
    }

}
```



Un esempio concreto: implementazione della gerarchia

```
public class Studente extends Persona{
    public Studente(String nome,
                    String cognome,
                       int eta) {
        super(nome,cognome,eta);
    }
}
```



Un esempio concreto: l'implementazione dell'archivio

```
import java.util.Vector;
public class Archivio1 {
    protected Vector persone;

    public Archivio1() {
        persone = new Vector(10);
    }

    public void aggiungi(Persona p){
        persone.add(p);
    }

    public void rimuovi(Persona p){
        persone.remove(p);
    }
```

Un esempio concreto: l'implementazione dell'archivio

```
public Persona get(int index){
        return (Persona)persone.get(index);
    }

    public int size(){
        return persone.size();
    }

}   // fine della classe Archivio1
```

Il problema di `Archivio1`

- La classe `Archivio1` gestisce (con una sola logica) tutti i tipi `Persona`; ciò può provocare type-unsafety
- In altre parole non vi è nessun meccanismo che controlli il tipo (specifico) dei dati inseriti/rimossi dall'archivio

Cosa ancora più importante: le incoerenze non possono essere rilevate a tempo di compilazione !

(si ricordi che `Studente` e `Professore` sono sottoclassi di `Persona`)

Un utilizzo scorretto

```
 public static void main(String argv[]){
    // creazione di studenti e professori
    Studente s1 = new Studente("Luca","Ferrari",26);
    Studente s2 = new Studente("Santi","Caballe",29);
    Studente s3 = new Studente("James","Gosling",50);
    Professore pr1 = new
Professore("Silvia","Rossi",27);
    Professore pr2 = new Professore("Simon","Ritter",
40);

    // creazione di due archivi separati
    Archivio1 archivio_prof = new Archivio1();
    Archivio1 archivio_stud = new Archivio1();
```

Un utilizzo scorretto

```
// aggiungo i prof e gli studenti ai relativi archivi
        archivio_prof.aggiungi(pr1);
        archivio_prof.aggiungi(pr2);
        archivio_prof.aggiungi(s1);

        archivio_stud.aggiungi(s1);
        archivio_stud.aggiungi(s2);
        archivio_stud.aggiungi(s3);
        archivio_stud.aggiungi(pr1);
```

Disastro imminente!

Disastro imminente!

Oggetti di tipo incoerente sono stati aggiunti agli archivi. Il compilatore e il sistema run-time non possono rilevare questo errore di logica, essendo l'archivio basato sulla superclasse dei tipi realmente utilizzati.

Un utilizzo scorretto

```
// stampa professori
for(int i=0;i<archivio_prof.size();i++){
  Professore pTemp =
  (Professore)archivio_prof.get(i);
  System.out.println("Professore "+i+" "+pTemp);
}

// stampa studenti
for(int i=0;i<archivio_stud.size();i++){
  Studente sTemp =
   (Studente)archivio_stud.get(i);
  System.out.println("Studente "+i+" "+sTemp);
      }
   }    // fine del metodo main
```

Un utilizzo scorretto

output di esecuzione

```
Professore 0 Silvia Rossi 27 – professore
Professore 1 Simon Ritter 40 - professore
Exception in thread "main“ java.lang.ClassCastException:
 seminario_20.generics.Studente
      at seminario_20.generics.Main1.main(Main1.java:32)
```

- **Il problema risiede nel cast fatto al momento dell'estrazione dall'archivio:**

```
Professore pTemp = (Professore)archivio_prof.get(i);
```

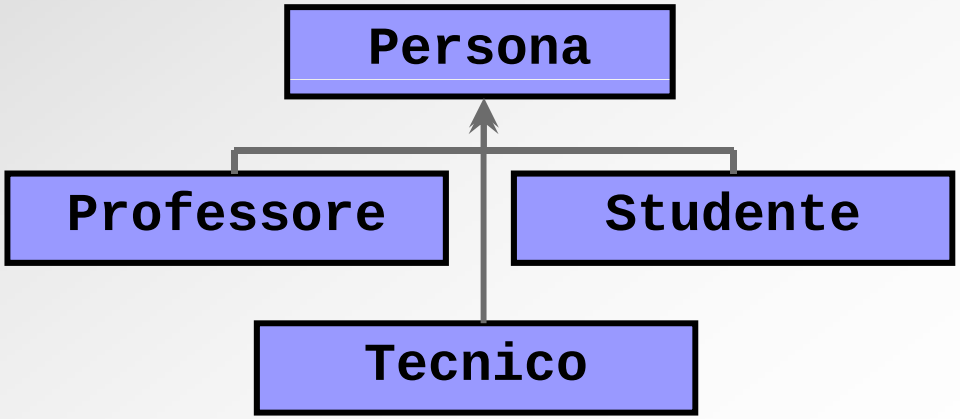
- **L'assunzione è corretta: l'archivio dei professori (studenti) dovrebbe contenere solo professori (studenti), quindi il cast esplicito è lecito**

Possibili soluzioni (in ordine di disperazione)

1. **realizzare una forma di archivio specifica per ogni tipo di dato presente**

   Ad esempio, realizzando un archivio che accetti come dato un `Professore`, non si potrà correre il rischio di inserirvi uno `Studente`

Questa soluzione risulta valida nel breve termine, ma non accettabile nel lungo periodo, essendo incapace di gestire correttamente il code refactoring. Ogni volta che un nuovo tipo viene inserito, un nuovo archivio deve essere implementato



Possibili soluzioni (in ordine di disperazione)

2. utilizzare interfacce differenti per ogni archivio

Definendo una interfaccia per ogni archivio, e “costringendo” l’archivio ad implementarle entrambe, si ottiene una soluzione molto Object Oriented, abbastanza sicura e sicuramente flessibile, al costo di una minima aggiunta di codice.

Possibili soluzioni (in ordine di disperazione)

2. utilizzare interfacce differenti per ogni archivio

```
public inteface archivio_studenti{
      public void aggiungi(Studente s);
      ...
}

public interface archivio_professori{
      public void aggiungi(Professore p);
}

public class Archivio
implements archivio_studenti, archivio_professori
{...}
```

Possibili soluzioni (in ordine di disperazione)

2. utilizzare interfacce differenti per ogni archivio

- I problemi di questa soluzione:

a) è potenzialmente non type-safe, essendo comunque possibile scavalcare le interfacce per lavorare direttamente sull'archivio

b) come per la soluzione precedente, ogni volta che un nuovo tipo viene aggiunto, l'archivio deve essere modificato di conseguenza

Possibili soluzioni (in ordine di disperazione)

3. **fornire un parametro all'archivio che imponga il tipo di istanza da gestire**

```
public Archivio1(Class clazz){
       this.managedType = clazz;
}

public void aggiungi_persona(Persona p) throws Exception{
       if( managedType != null &&
           (p.getClass()!= managedType )  ){
             throw new Exception("Tipo sbagliato");
       }
       // inserimento dell'argomento…
}
```

Anticipa ciò che avviene a run-time

per un cast incorretto!

Possibili soluzioni (in ordine di disperazione)

4. eliminare la gerarchia

- Non è una soluzione Object Oriented!
- Produce la scrittura di molto codice in più

Generics

- **È ora possibile un controllo accurato sui tipi a tempo di compilazione**
- **ma le strutture dati devono essere progettate come *tipi parametrici* (*generics*)**
- **Una classe che sfrutti generics non ha un tipo di dato predefinito, ma lo riceve a tempo di istanziazione**

Generics

- In altre parole, con generics, è possibile costruire classi con algoritmi standard e capaci di agire su più tipi di dati, ma in modo coerente

- "From the perspective of both software engineering and programming pedagogy, Java has a crude type system. Its most significant failing is the lack of support for generic types."

  (Erica Allen, Safe Instantiation in Generic Java, PPPJ 2004)

- La sintassi di generics fa uso delle parentesi angolari (<,>) per indicare un tipo da specificare in seguito

Usando generics…

```
import java.util.Vector;
public class Archivio2<E> {
   protected Vector persone;

    public Archivio2() {
        persone = new Vector(10);
    }

    public void aggiungi(E p){
        persone.add(p);
    }

    public void rimuovi(E p){
        persone.remove(p);
    }
```

Con questa dichiarazione si indica che l'identificatore E fa riferimento ad un tipo di dato che sarà specificato nel codice che userà Archivio2

I metodi ora fanno riferimento a variabili di tipo E

Usando generics…

```
    public E get(int index){
        return (E)persone.get(index);
    }

    public int size(){
        return persone.size();
    }

}   // fine della classe Archivio2
```

Usando generics...

```
public static void main(String argv[]){
    ...
     Archivio2<Professore> archivio_prof =
                           new Archivio2<Professore>();
     Archivio2<Studente> archivio_stud =
                           new Archivio2<Studente>();
     Archivio2<Persona>  archivio_per =
                           new Archivio2<Persona>();

     archivio_prof.aggiungi(pr1);
     archivio_prof.aggiungi(pr2);
     archivio_prof.aggiungi(s1);
   ...
}
```

output di compilazione

```
Main2.java:34: aggiungi(seminario_20.generics.Studente) in
seminario_20.generics.Archivio2<seminario_20.generics.Studente>
cannot be applied to (seminario_20.generics.Professore)
```

I vantaggi di generics

- Le incoerenze di tipo sono rilevate a tempo di compilazione, e non a tempo di esecuzione, aiutando il programmatore nel trovare errori di logica
- Si noti che generics non impedisce di usare la gerarchia mischiando i tipi, ma semplicemente richiede che si sia coscienti di ciò che si sta facendo:

```
Archivio2<Persona>  archivio_per =
                              new Archivio2<Persona>();
```

Wildcards

- Generics ammette l'uso del carattere '?' come speciale wildcard, con i seguenti significati:
- `<?>` indica tutti i tipi della classe cui si fa riferimento. Ad esempio `Archivio2<?>` indica sia `Archivio2<Studente>` che `Archivio2<Professore>`
- `<? extends type>` indica tutti i tipi che ereditano da `type`. Ad esempio `Archivio2<? extends Persona>` indica tutti i tipi di `Archivio2` parametrizzati da `Persona`
- `<? super type>` simile al caso precedente, ma tratta superclassi

Specializzare una classe con generics

- **È possibile ereditare da una classe e aggiungere, nel contempo, il supporto a generics, facendo però attenzione affinché i metodi non siano in conflitto**

```
public class Archivio3<E> extends Archivio1{

      public void aggiungiElement(E p){

        persone.add(p);

      }

    ...

}
```

ATTENZIONE: si ricordi che Studente (Professore) è anche Persona, quindi un metodo aggungi(E p) potrebbe andare in conflitto con Archivio1#aggiungi(Persona p) qualora il tipo sia ancora una Persona.

Cosa c’è dietro a generics

- La “magia” di generics risiede nel nuovo sistema di compilazione
- Il compilatore effettua ora alcuni passi di manipolazione sintattica (*type erasurers*) al fine di forzare eventuali errori di casting

Generics in azione: codice utilizzato

- Il tag di generics viene rimosso, e il codice viene compilato sostituendo al tipo parametrico `Object`

```
public class Archivio2<E> {
    protected Vector persone;

     public Archivio2() {
         persone = new Vector(10);
     }

     public void aggiungi(Object p){
         persone.add(p);
     }

     public void rimuovi(Object p){
         persone.remove(p);
     }
```

Generics in azione: codice utilizzatore

- Il compilatore rimuove i tag di generics, e "forza" dei cast

```
public static void main(String argv[]){
    ...

    Archivio2<Professore> archivio_prof =
                          new Archivio2<Professore>();
    Archivio2<Studente> archivio_stud =
                          new Archivio2<Studente>();
    Archivio2<Persona>  archivio_per =
                          new Archivio2<Persona>();

    archivio_prof.aggiungi((Professore)pr1);
    archivio_prof.aggiungi((Professore)pr2);
    archivio_prof.aggiungi((Professore)s1);
   ...
}
```

Generics e il resto del mondo

- La libreria Java supporta appieno generics:

```
public static void main(String argv[]){
        Vector<String> vettore = new Vector<String>();
        for(int i=0;i<10;i++){
            vettore.add(new String("stringa n."+i));
        }
        for(String s: vettore){
            System.out.println(s);
        }
    }
```

Quando usato con generics, foreach lavora sul tipo di dato corretto!

Ma non tutto è generics!

- **Se si tenta di utilizzare una classe "normale" come fosse generics, si ottiene un errore di compilazione.**

```
Archivio1<Professore> = new Archivio1<Professore>();
```

output di compilazione

```
Main4.java:15: not a statement
  Archivio1<Professore> = new Archivio1<Professore>();
Main4.java:15: ';' expected
  Archivio1<Professore> = new Archivio1<Professore>();
```

Il rovescio della medaglia

- **Generics consente di usare un algoritmo generico in *type-safe*, ma per impostazione predefinita non impedisce di usare l'algoritmo per istanze diverse da quelle per cui questo è stato progettato (cosa impedita dalla conformità)!**

```
Archivio2<String> = new Archivio2<String>();
```

- **La ragione di ciò risiede nel modo in cui la classe sottoposta a generics viene compilata: tutti gli identificatori sono sostituiti con `Object`.**

È possibile limitare i tipi utilizzabili!

Limitare l'uso dei tipi

```
public class Archivio2<E extends Persona>{…}
```

Se a questo punto si tenta di creare un archivio con un tipo sbagliato, si ottiene un errore di compilazione

```
Archivio2<String> archivio_stud =
                        new Archivio2<String>();
```

output di compilazione

```
seminario_20\generics\Main9.java:13: type parameter
java.lang.String is not within its bound
        Archivio2<String> archivio_stud =
                                  new Archivio2<String>();
```

Generics non implica relazioni!

- **Tutte le istanze create in modo parametrizzato condividono la stessa classe**

```
Archivio2<Persona> aPersona = …
Archivio2<Studente> aStudente = …
```

Non sono in relazione!

(anche se `Studente` eredita da `Persona`)

Generics extends Generics

- È possibile estendere una classe che fa uso di generics, la sottoclasse può a sua volta fare uso di generics
- Valgono tutte le regole dell'ereditarietà (es. overriding)!
- Esempio: estendere l'archivio visto in precedenza (`Archivio2`) in modo che possa memorizzare associazioni `Studente-Professore` in type-safe

Esempio

```
import java.util.Hashtable;

public class Archivio5<E,R> extends Archivio2<E>{
    // relazioni
    protected Hashtable relazioni;

    public Archivio5(){
    super();
    relazioni = new Hashtable();
    }

    public void aggiungiRelazione(E p1, R p2){
    relazioni.put(p1,p2);
    }
```

Esempio

```
public void stampaRelazioni(){
    Enumeration<E> chiavi = relazioni.keys();

    while(chiavi.hasMoreElements()){
        E chiave = chiavi.nextElement();
        System.out.println("Relazione "+chiave+" - “

    +relazioni.get(chiave));
    }
    }

}
```

Esempio: utilizzo

```
public class Main5{
    public static void main(String argv[]){
        Studente s1 = new
Studente("Luca","Ferrari",26);
        Studente s2 = new
Studente("Santi","Caballe",29);
        Studente s3 = new
Studente("James","Gosling",50);

        Professore pr1 = new
Professore("Silvia","Rossi",37);
        Professore pr2 =
                    new Professore("S.","Ritter", 40);
```

Esempio: utilizzo

```
    Archivio5<Studente,Professore> archivio =
              new Archivio5<Studente,Professore>();

    archivio.aggiungi(s1);
    archivio.aggiungi(s2);
    archivio.aggiungi(s3);

    archivio.aggiungiRelazione(s1,pr1);
    archivio.aggiungiRelazione(s2,pr2);

    archivio.stampaRelazioni();
  }
}
```

output di esecuzione

```
Relazione Luca Ferrari 26 - Silvia Rossi 37 - professore
Relazione Santi Caballe 29 – S. Ritter 40 - professore
```

Considerazioni sull'esempio

- È possibile utilizzare generics con più di un tipo di parametro:

```
public class Archivio5<E,R> extends Archivio2<E>{...}
Archivio5<Studente,Professore> archivio = new
     Archivio5<Studente,Professore>();
```

- Il type-safe è garantito:

```
archivio.aggiungi(pr1);
archivio.aggiungiRelazione(pr1,s1);
```

output di compilazione

```
seminario_20/generics/Main5.java:17: aggiungi(seminario_20.generics.Studente) in
seminario_20.generics.Archivio2<seminario_20.generics.Studente> cannot be applied
to (seminario_20.generics.Professore)
        archivio.aggiungi(pr1);
                 ^
seminario_20/generics/Main5.java:20:
aggiungiRelazione(seminario_20.generics.Studente,seminario_20.generics.Professore)
in
seminario_20.generics.Archivio5<seminario_20.generics.Studente,seminario_20.generi
cs.Professore> cannot be applied to
(seminario_20.generics.Professore,seminario_20.generics.Studente)
        archivio.aggiungiRelazione(pr1,s1);
```

Templates? No grazie!

- Anche se molto simili nella sintassi e nell'utilizzo, i Java generics non sono la stessa cosa dei template C++
- I template C++ si riconducono a macro del preprocessore, che producono il codice sorgente di una nuova classe con i tipi "fissati"
- Generics opera a livello di compilatore e non "sporca" il codice della classe che si sta utilizzando

Generics è un modo di concepire e scrivere software!

Ergonomics & performance

Ergonomics

- La JVM include ora una serie di opzioni per il tuning e il profiling delle applicazioni
- Le opzioni di ergonomics coprono la scelta del garbage collector, l'utilizzo del nuovo sistema di class-sharing, dimensione dell'heap e degli spazi generazionali, ecc

La scelta del garbage collector

Sono disponibili le seguenti opzioni:

- `-Xnoclassgc` disabilita la class garbage collection
- `-Xincgc` (mark & sweep) abilita il garbage collector incrementale
- `-XX:+UseParallelGC` abilita il GC generazionale parallelo (throughput collector)
- `-XX:+UseConcMarkSweepGC` versione parallela di `–Xincgc`

Il garbage collector in azione

- Usando il flag –verbose:gc è possibile osservare il garbage collector in azione:

La dimensione della memoria

- È possibile un controllo accurato sulla dimensione della memoria heap:
  - `-Xms` imposta la dimensione iniziale dell'heap
  - `-Xmx` imposta la dimensione massima dell'heap
  - `-Xss` imposta la dimensione dello stack di un thread
  - `-XX:MinHeapFreeRatio,`

`-XX:MaxHeapFreeRatio` impostano il limite inferiore/superiore (in percentuale) dell'heap libero per forzare un passagio del GC. Per un architettura a 32 bit i valori sono rispettivamente 40% e 70%

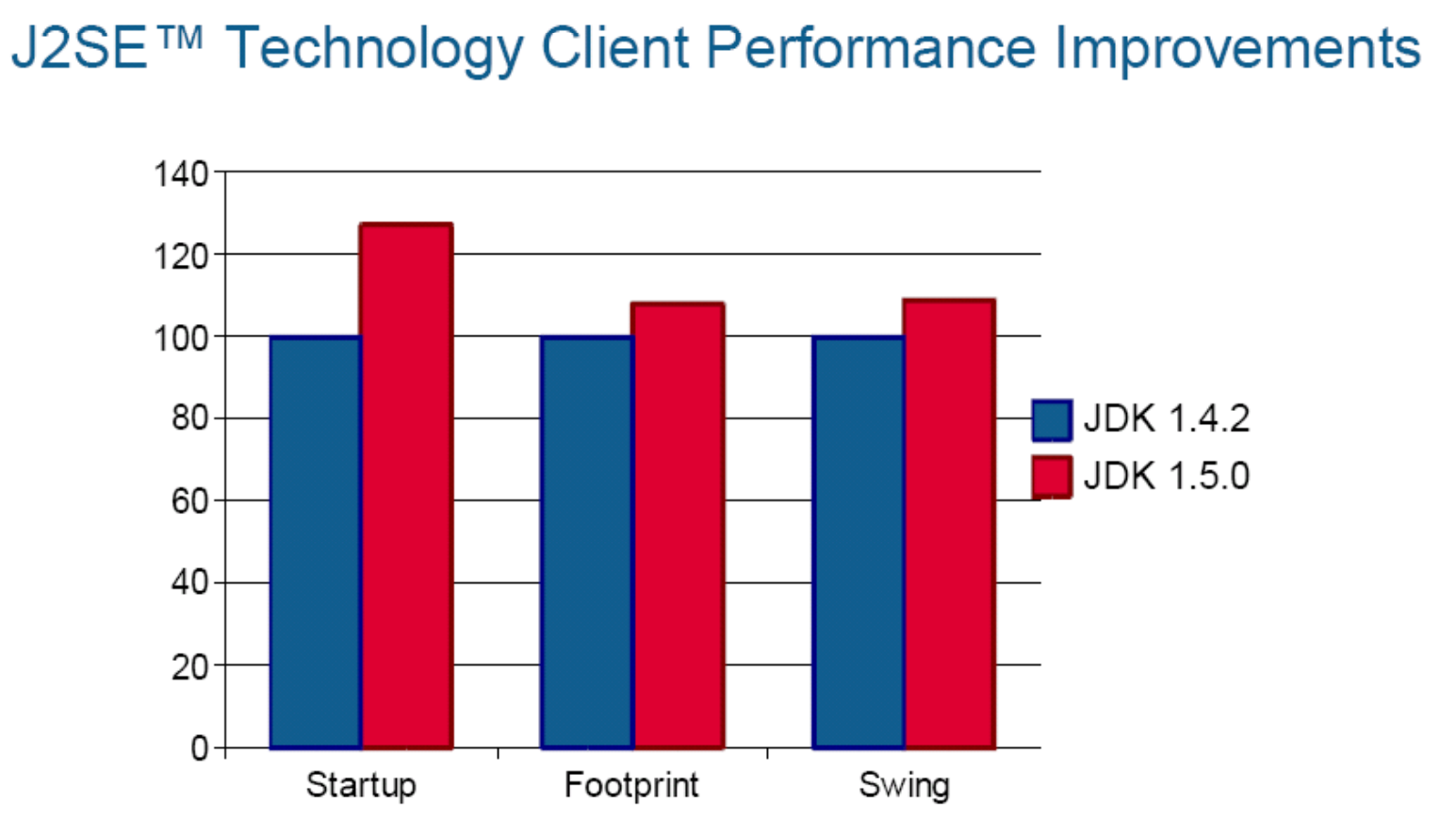
Un esempio casalingo

```
public class runner{
  public static void main(String argv[]){
    long start,stop;
    start = System.currentTimeMillis();
    for(int i=0;i<10000;i++){
        for(int j=0;j<10000;j++){
           String s = new String("Una stringa
                              abbastanza lunga!!!!!!");
           }
         JFrame f = new JFrame("Nuovo frame");
    }

    stop = System.currentTimeMillis();
    System.out.println("Tempo di esecuzione “
                                +(stop-start)+" ms");
  }
}
```

Un esempio casalingo

normale	12518 ms
`-noclassgc`	12478 ms
`-incgc`	9584 ms
`-int`	47218 ms
`-Xms 64000000`	9033 ms
`-Xms 64000000 –XX:MinHeapFreeRatio=1`	8843 ms
`-Xms 64000000 –XX:MinHeapFreeRatio=1 -XX:MaxHeapFreeRatio=99`	8682 ms

Prestazioni del JDK 1.5



Prestazioni del JDK 1.5